Standard ML Weak Polymorphism Can Be Sound

John Greiner

May, 1993

CMU-CS-93-160

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Adding ML-style references to a Hindley-Milner polymorphic type system is troublesome because such a system is unsound with naive polymorphic generalization of reference types. Tofte [12] introduced a distinction between *imperative* and *applicative* type variables, such that applicative type variables are never in reference types, that provides a simple static analysis of which type variables may be polymorphically generalized. MacQueen's [7] *weak* type variables generalize imperative type variables with a counter called a *strength*. The finer distinction allows a more accurate analysis of when a reference may be created, and thus which type variables may be generalized.

Unfortunately, weak polymorphism has been presented only as part of the implementation of the SML/NJ compiler, not as a formal type system. As a result, it is not well understood, as its more subtle points are not well known. Furthermore, while versions of the implementation have repeatedly been shown unsound, the concept has not been proven sound or unsound. We present several formal systems of weak polymorphism, show their connection to the SML/NJ implementation, and show the soundness of most of these systems.

This research was sponsored in part by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168, and in part by the ONR Graduate Fellowship Program.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: Data types and structures, lambda calculus and related systems, dynamic storage management, functional constructs

1. Background

A reference cell is an assignable memory location and is a primary imperative feature of Standard ML of New Jersey (SML/NJ). The language is unsound if reference types are polymorphically generalized in the usual manner of Hindley-Milner type system. For example, a list of integers could be stored in the cell, and a list of booleans read from it, as in the expression

```
let val a = ref nil in a := [1]; not (hd(!a)) end
```

While it is unsound to polymorphically generalize types of the form α *ref*, generalizing function types involving references is not necessarily so. A simple example is

```
let val ref' = fn x => ref x in (ref' 1,ref' true) end
```

A number of type systems have been proposed to allow code such as this while preserving soundness [1, 4, 6, 10, 11, 13]. Of particular interest for this paper are those of Tofte [12] and MacQueen [7].

In the standard Hindley-Milner type system, generalization is allowed on all free type variables not occurring in the *(variable) type assumption*, a mapping from variables to type schemes. However, with references, it is also necessary to have a *location type assumption*, assigning a type to each location of the store [12]. The unsoundness of the naive static semantics is a result of generalizing type variables occurring free in the location type assumption [12]. Since neither the store nor the location type assumption can be known statically, a safe approximation must be made to determine those type variables which may occur in the location type assumption and therefore should not be generalized [12].

In order to provide such a safe approximation, Tofte introduces a distinction between two classes of type variables, called *applicative* and *imperative*. Imperative type variables are used to statically track values that may be placed in reference cells. Applicative type variables are not used with reference types and can always be polymorphically generalized. Additionally, imperative type variables can be generalized if the evaluation of the *let*-bound expression does not lead to creation of a reference cell. Since this is undecidable, Tofte defines an expression to be *non-expansive* if it is syntactically a value other than a reference cell. The second example above is then type correct. Since functions are non-expansive, the local type assumption maps `ref'` to the type scheme ∀ `u.u -> u ref`, where `u` is imperative, which instantiates to both `int -> int ref` and `bool -> bool ref`. The first example is still rejected as desired since the expression `ref true` is not non-expansive, and its type is not generalized.

However, this static analysis is overly conservative. In the example

```
let val ref2 = fn x => fn y => ref x
in
    let val ref1 = ref2 nil
    in
        (true :: !(ref1 ()),1 :: !(ref1 ()))
    end
end
```

the type of `ref1` is not generalized because application expressions are considered expansive. Since there is no generalization, the expression does not have a type since `ref1` cannot be of type `unit -> bool ref` and `unit -> int ref`. However, it is sound to allow such code since a different reference cell is created for each call.

One method to improve upon Tofte's system is to track not only which values may be placed in reference cells, but *when* the cells are created. This additional information can then be used for a more accurate definition of non-expansiveness. This is the essence of MacQueen's *weak* polymorphic types

2. Weak Polymorphic Types

Weak polymorphism expands on Tofte's distinction of type variables. To produce a better static analysis of what values may be in reference cells, type variables are indexed by an integer, known equivalently as its *strength* or *weakness*. A strength s of a type variable in the type of an expression indicates that during the evaluation of the expression supplied with less than s arguments, no cell is created of a type involving that type variable. Applicative type variables correspond to those of infinite strength, whereas those of finite strength are *weakly polymorphic*. In particular, a type variable with strength of zero corresponds to the possible creation of a cell with a type involving that type variable during the evaluation of the expression.

Non-critical type variables, those of positive strength, are generalized, but *critical* variables are not. Thus the improvement over Tofte's system stems from being able to generalize some *imperative* or *weak* variables. This is similar to Tofte's non-expansive condition for allowing generalization. A more in-depth comparison of the two related systems in found in Section 7.

Using such motivation, we can develop the basic ideas of weak types. Since a reference cell must have a critical type,

```
    ref nil : '0a list ref
```

and purely functional terms have types of infinite strength.[1]

```
    fn x => x : 'a -> 'a
```

Abstraction increments strengths, since they count the number of applications until a reference is created.

```
    fn x => ref nil : 'b -> '1a ref
    fn x => ref x : '1a -> '1a ref
```

Similarly, application decrements the strengths in the function position.

```
    (fn x => ref x) nil : '0a list ref
```

If the argument of an application has a weak type, the analysis must make a conservative approximation. In general, the function may in turn apply its argument to multiple arguments, where each application corresponds to a decrement in strengths. Statically, the conservative assumption is made that enough applications are performed for a reference cell to be produced. For example, the strength of 'a in

```
    (fn x => fn y => ref x) : '2a -> 'b -> '2a ref
```

must be made critical when the expression is used as an argument in

```
    (fn f => f nil nil) (fn x => fn y => ref x) : '0a list ref
```

Following the previous examples, the following code would be assigned a type with negative strength

```
    ((fn f => f nil) (fn x => fn y => ref x)) nil : '~1a list ref
```

since the application to `nil` decrements the already critical strengths of the function. SML/NJ avoids negative strengths at the top-level in most cases. But in Version 0.66, which follows this motivation closely,

```
    (let val x = ref (fn z => z) in fn y => x end) () : ('~1a -> '~1a) ref
```

Intuitively, the *let* expression has type `unit -> ('0a -> '0a) ref` since a reference is created, and the application decrements the strengths one more. More commonly, negative strengths are only used when type checking sub-expressions of the original expression as in

[1] In SML/NJ, infinite strengths are not printed.

```
ref (fn z => z) : ('0a -> '0a) ref
```

The strength of `'a` when type checking `z` can be thought of as being `~1`, which is then incremented by the abstraction.

Weak polymorphism has been developed by MacQueen within the type inference algorithm of SML/NJ. Only this algorithm has served as the definition of the type system, and unfortunately, numerous implementations have been shown to be unsound. Each has had problems which could be ascribed to implementation details, but the concept has not been proven sound. Additionally, key ideas of the algorithm, such as this conservative approximation at application, have not been widely known, so the system has been poorly understood even by skilled SML/NJ programmers.

This paper addresses these problems. In Section 3, this motivation is transformed into a formalism, the soundness of which is outlined in Section 4 and given in full in the Appendix. In Section 5, it is shown how this formalism relates to a more algorithmic formalism, which is also sound. Other details of a comparison to the algorithm of SML/NJ are presented in Section 6, and a comparison to other approaches is in Section 7.

3. A Declarative Formalism – $\lambda\Sigma$

This section presents a formal definition of an ML-like language, related to the previous motivation of weak polymorphism. This includes the rules and motivation for the syntax, dynamic semantics, and static semantics, as well as the notation used in the semantics and proof.

The expression language used is defined by

$$\begin{array}{rcl} x \in \mathit{variables} & & \\ l \in \mathit{locations} & & \\ e \in \mathit{expressions} & ::= & x \mid l \mid () \mid \mathit{ref}\, e \mid !e \mid e_1{:=}e_2 \mid \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 \mid \mathit{fn}\ x \Rightarrow e \\ v \in \mathit{values} & ::= & l \mid () \mid \mathit{fn}\ x \Rightarrow e \end{array}$$

The free variables of the expression, $FV(e)$, are defined in the usual manner. Capture-avoiding expression substitution is denoted $[e'/x]e$.

An empty mapping is denoted by $\cdot$, while the extension of a mapping over an additional domain element is denoted like $X[d \mapsto r]$.[2] Mappings are also abbreviated like $[d_1 \mapsto r_1, \ldots, d_n \mapsto d_n]$. The union of disjoint mappings is written as X, X'.

Dynamic Semantics

The dynamic semantics is defined by the following standard rules, where a memory μ is a finite mapping between locations and values. The judgment $\mu \vdash e \Longrightarrow v, \mu'$ is well-formed if e is closed with respect to all variables, and reads "Given the memory μ, the expression e evaluates to v, resulting in a new memory μ'."

$$\mu \vdash v \Longrightarrow v, \mu \qquad \text{(VAL)}$$

$$\frac{\mu \vdash e \Longrightarrow v, \mu'}{\mu \vdash \mathit{ref}\, e \Longrightarrow l, \mu'[l \mapsto v]} \quad \mathit{if}\ l \notin \mathit{dom}(\mu') \qquad \text{(ALLOC)}$$

[2] Except for type assumptions, written like $\Gamma[x : \tau]$

$$\frac{\mu \vdash e \Longrightarrow l, \mu'}{\mu \vdash !e \Longrightarrow \mu'(l), \mu'} \qquad \text{(CONT)}$$

$$\frac{\mu \vdash e_1 \Longrightarrow l, \mu_1 \qquad \mu_1 \vdash e_2 \Longrightarrow v, \mu_2[l \mapsto v']}{\mu \vdash e_1{:=}e_2 \Longrightarrow (), \mu_2[l \mapsto v]} \qquad \text{(UPD)}$$

$$\frac{\mu \vdash e_1 \Longrightarrow \mathit{fn}\ x \Rightarrow e'_1, \mu_1 \qquad \mu_1 \vdash e_2 \Longrightarrow v_2, \mu_2 \qquad \mu_2 \vdash [v_2/x]e'_1 \Longrightarrow v, \mu'}{\mu \vdash e_1 e_2 \Longrightarrow v, \mu'} \qquad \text{(APPLY)}$$

$$\frac{\mu \vdash e_1 \Longrightarrow v_1, \mu_1 \qquad \mu_1 \vdash [v_1/x]e_2 \Longrightarrow v_2, \mu_2}{\mu \vdash \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 \Longrightarrow v_2, \mu_2} \qquad \text{(BIND)}$$

Static Semantics

The types and type schemes are defined by

$$\begin{array}{rcl} s, a, w \in \textit{strengths} & = & \mathbf{Z} \cup \{\infty\} \\ \alpha, \beta \in \textit{tyvars} & & \\ \tau \in \textit{types} & ::= & \alpha \mid \textit{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \textit{ref} \\ \sigma \in \textit{typeschemes} & ::= & \forall \Sigma . \tau \end{array}$$

where a *strength context* Σ is a finite mapping from type variables to strengths. The trivial type scheme $\forall . . \tau$ is abbreviated by τ. Since all instances of a type variable in a type or type scheme must have the same strength, strength contexts are used to maintain consistency.

The free (bound) type variables of a type or type scheme are denoted by $FTV(\tau)$ ($BTV(\tau)$). These functions are also extended to location and variable type assumptions, and are also extended to be n-ary functions so that, for example, $FTV(\Lambda, \Gamma) = FTV(\Lambda) \cup FTV(\Gamma)$.

Type variables of non-positive strength are *critical*, or $Crit_\Sigma(A)$, if for all $\alpha \in A$, $\Sigma(\alpha) \leq 0$. Similarly, $NonCrit_\Sigma(A)$, if for all $\alpha \in A$, $\Sigma(\alpha) > 0$. A type is (non-)critical relative to Σ if all of its type variables are (non-)critical in Σ.

$Weaker_\Sigma(A, s)$ holds if for all $\alpha \in A$, $\Sigma(\alpha) = \infty$ or $\Sigma(\alpha) \leq s$, and similarly, $SWeaker_\Sigma(A, s)$ holds if for all $\alpha \in A$, $\Sigma(\alpha) \leq s$. Let $(\Sigma + c)(\alpha) = \Sigma(\alpha) + c$. The minimum of two strength contexts having the same domain, $\min(\Sigma_1, \Sigma_2)$, is defined by the point-wise minimum.

A strength context Σ' is *weaker (below s)* than strength context Σ of the same domain, $\Sigma' \leq_s \Sigma$, if for all $\alpha \in dom(\Sigma')$, $\Sigma'(\alpha) = \Sigma(\alpha)$ whenever $\Sigma(\alpha) \geq s$, and otherwise $\Sigma'(\alpha) \leq \Sigma(\alpha)$. Of particular interest is the case when $s = 1$, which implies that the two strength contexts agree on all positive strengths, but that Σ' is *more critical* than Σ. Note that $\Sigma' + 1 \leq_s \Sigma + 1$ implies $\Sigma' \leq_s \Sigma$, which implies $\Sigma' \leq_{s+1} \Sigma$, but the converses do not hold.

Instantiation is defined much as it is in the standard Hindley-Milner type system, but with restrictions on weak type variables. A type scheme *instantiates* to a type,

$$\vdash_\Sigma \forall[\alpha_1 \mapsto s_1, \ldots, \alpha_n \mapsto s_n].\tau \succeq \tau'$$

if there exists a type substitution $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ such that $S(\tau) = \tau'$, and for all $\alpha \in FTV(\tau_i)$, $\Sigma(\alpha) \leq s_i$. Similarly, relative to Σ, a type scheme σ' is *more general* than σ, written $\vdash_\Sigma \sigma' \succeq \sigma$, if for any type τ such that $\vdash_\Sigma \sigma \succeq \tau$, then $\vdash_\Sigma \sigma' \succeq \tau$. For example, for any $j < i$,

$$\vdash_. \forall[\alpha \mapsto i].\alpha \rightarrow \alpha \succeq \forall[\alpha \mapsto j].\alpha \rightarrow \alpha$$

but *not*

$$\vdash_{\cdot} \forall[\alpha \mapsto j].\alpha{\rightarrow}\alpha \succeq \forall[\alpha \mapsto i].\alpha{\rightarrow}\alpha$$

and the type schemes $\forall[\alpha \mapsto j, \beta \mapsto i].\alpha{\rightarrow}\beta$ and $\forall[\alpha \mapsto i, \beta \mapsto j].\alpha{\rightarrow}\beta$ are incomparable.

A type judgment $\Lambda;\Gamma \vdash_\Sigma e : \tau$ reads "With strength context Σ, given the location type assignment Λ and variable type assignment Γ, e has type τ." Such a judgment is well-formed if $dom(\Sigma) \supseteq FTV(\Lambda, \Gamma, \tau)$ and this relation. Derivability is defined by the following rules which use the definitions in the remainder of this section.

$$\frac{\vdash_\Sigma \Gamma(x) \succeq \tau}{\Lambda;\Gamma \vdash_\Sigma x : \tau} \qquad (\text{VAR}_\text{D})$$

$$\frac{\Lambda(l) = \tau}{\Lambda;\Gamma \vdash_\Sigma l : \tau\ ref} \qquad (\text{LOC}_\text{D})$$

$$\Lambda;\Gamma \vdash_\Sigma () : unit \qquad (\text{UNIT}_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_\Sigma e : \tau}{\Lambda;\Gamma \vdash_\Sigma ref\, e : \tau\ ref} \ \ if\ Crit_\Sigma(FTV(\tau)) \qquad (\text{REF}_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_\Sigma e : \tau\ ref}{\Lambda;\Gamma \vdash_\Sigma !e : \tau} \qquad (!_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_\Sigma e_1 : \tau\ ref \qquad \Lambda;\Gamma \vdash_\Sigma e_2 : \tau}{\Lambda;\Gamma \vdash_\Sigma e_1{:=}e_2 : unit} \qquad (:=_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_{\Sigma+1} e_1 : \tau_2{\rightarrow}\tau \qquad \Lambda;\Gamma \vdash_\Sigma e_2 : \tau_2}{\Lambda;\Gamma \vdash_\Sigma e_1\ e_2 : \tau} \ \ if\ Weaker_\Sigma(FTV(\tau_2), 0) \qquad (\text{APP}_\text{D})$$

$$\frac{\Lambda;\Gamma[x : \tau_1] \vdash_{\Sigma-1} e : \tau_2}{\Lambda;\Gamma \vdash_\Sigma fn\ x \Rightarrow e : \tau_1{\rightarrow}\tau_2} \ \ if\ x \notin dom(\Gamma) \qquad (\text{LAM}_\text{D})$$

$$\frac{\begin{array}{c}\Lambda;\Gamma \vdash_{\Sigma,\Sigma'} e_1 : \tau_1 \\ \Lambda;\Gamma[x : \forall\Sigma'.\tau_1] \vdash_\Sigma e_2 : \tau_2\end{array}}{\Lambda;\Gamma \vdash_\Sigma let\ x{=}e_1\ in\ e_2 : \tau_2} \ \ if\ \begin{array}{l} x \notin dom(\Gamma),\ NonCrit_{\Sigma'}(dom(\Sigma')),\ and \\ Weaker_\Sigma((FTV(\tau_1) \cap dom(\Sigma)) - FTV(\Lambda,\Gamma), 0)\end{array} \qquad (\text{LET}_\text{D})$$

Note that the strengths are incremented and decremented in (APP_D) and (LAM_D). The side condition of (APP_D) enforces the conservative approximation previously described. The side condition on (REF_D) simply reflects the usual treatment of *ref* as a functional primitive having the type scheme $\forall[\alpha \mapsto 1].\alpha{\rightarrow}\alpha\ ref$.

In (LET_D), Σ and Σ' have disjoint domains by the definition of the map union operator. So, by the well-formedness of the second precondition, the domain of Σ' is disjoint from the free type variables of both type assumptions. By using the strength context Σ' only in the first subderivation, it is explicit that the names of the generalized type variables are relevant only locally.

The last side condition of (LET_D) states that any finitely positive type variables in τ_1, but not in the type assumptions, *must* be generalized. Without this restriction, the following judgment is derivable.

$$\bullet;\bullet \vdash_{[\alpha\mapsto 2]} \mathit{let}\ x{=}\mathit{fn}\ z \Rightarrow \mathit{fn}\ y \Rightarrow \mathit{ref}\ y\ \mathit{in}\ \ x() : \alpha \rightarrow \alpha\ \mathit{ref}$$

This is a very technical condition. It disallows the rule's use if any type variable in $FTV(\tau_1) \cap FTV(\tau_2) - FTV \Lambda, \Gamma$ is finitely positive in Σ. But, if this were the case, there exists another derivation to allow the *let* expression to type by renaming these type variables in τ_1 with fresh type variables, and adding them to Σ'.

To relate the locations in the dynamic and static semantics, a memory μ *type matches (with respect to* Σ*)* the type context Λ , or $\vdash_\Sigma \mu : \Lambda$, if $dom(\mu) = dom(\Lambda)$, if for all $l \in dom(\mu)$, $\Lambda;\bullet \vdash_\Sigma \mu(l) : \Lambda(l)$.

4. Soundness

This section presents an overview of the soundness proof of the formalism, including the statement and proof sketches of the theorem and main lemmas. For the full proof, refer to the Appendix. Soundness is shown by proving a form of type preservation under evaluation, as in [2, 3, 12, 15]. The theorem and lemmas are similar to those for the functional Hindley-Milner and Tofte's imperative type systems, except that extra conditions are needed to keep tight control on the strengths of pertinent type variables, and the notion of an occurrence adds many complications.

Unfortunately, while evaluation preserves types, it does not necessarily preserve strengths. For example, consider the following expressions:

$$\begin{aligned}
e_0 &= \mathit{fn}\ z \Rightarrow z \\
e_1 &= \mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow x e_0 e_0 \\
e_2 &= \mathit{fn}\ a \Rightarrow (\mathit{let}\ b{=}\mathit{ref}\ a\ \mathit{in}\ \mathit{fn}\ c \Rightarrow !b)() \\
e &= e_1 e_2 \\
v &= \mathit{fn}\ y \Rightarrow e_2 e_0 e_0
\end{aligned}$$

In particular, e_2 is a function which is assigned a critical type, and e evaluates to v. The strongest typings for e and v are

$$\bullet;\bullet \vdash_{[\alpha\mapsto 0,\beta\mapsto\infty]} e : \beta\rightarrow\alpha\rightarrow\alpha$$
$$\bullet;\bullet \vdash_{[\alpha\mapsto -1,\beta\mapsto\infty]} v : \beta\rightarrow\alpha\rightarrow\alpha$$

It is sound to assign the stronger type to v, but the analysis provided by the formalism is not able to give this stronger type. As will be proved, no such example exists for non-critical types.

This does not entirely contradict the usual intuition that if e evaluates to v, then v could be given a more general type than e. In most cases, v could be assigned the same or higher strengths. For a simple example, consider

$$\begin{aligned}
e &= (\mathit{fn}\ x \Rightarrow x)(\mathit{fn}\ y \Rightarrow \mathit{ref}\ y) \\
v &= \mathit{fn}\ y \Rightarrow \mathit{ref}\ y
\end{aligned}$$

Since $\bullet;\bullet \vdash_{[\alpha\mapsto 0]} e : \alpha\rightarrow\alpha\ \mathit{ref}$, the Type Preservation theorem states that there exists a critical strength s such that $\bullet;\bullet \vdash_{[\alpha\mapsto s]} v : \alpha\rightarrow\alpha\ \mathit{ref}$. But, in fact, v can be typed with the higher strength context $[\alpha \mapsto 1]$.

The Top-Level Type Preservation theorem states that if e evaluates to v, then e and v have the same type, although v may require more critical strengths. Furthermore, any cells created during this evaluation have critical types.

Theorem (Top-Level Type Preservation Under Evaluation) *If* $\mu \vdash e \Longrightarrow v, \mu'$, *and* $\bullet;\bullet \vdash_\Sigma e : \tau$, *then there exists* Λ_0, *and* $\Sigma_0 \leq_1 \Sigma$ *such that* $\Lambda_0;\bullet \vdash_{\Sigma_0} v : \tau$, $\vdash_{\Sigma_0} \mu' : \Lambda_0$, *and* $Crit_{\Sigma_0}(FTV(\Lambda_0))$.

Proof Sketch: This is proved by generalizing the theorem to all location type assumptions, and generali ing the strength context, and then by structural induction on the evaluation derivation. The (APPLY) and (BIND) cases require the following Value Substitution lemma. The last assumption of that lemma is achieved through the side conditions on the ($\mathrm{APP_D}$) and ($\mathrm{LET_D}$) rules. The rest of the cases are not difficult, although extensive use is made of the Weakening and Strengthening lemmas. □

The Value Substitution lemma states that, under restrictions, the type of an expression is stable under substitution of a value for a variable of more general type. The result of the substitution may require more critical strengths.

Lemma (Value Substitution) *If*

- $\Lambda; \cdot \vdash_{\Sigma,\Sigma_1} v : \tau_1$,
- $\Lambda; \Gamma[x : \forall\Sigma_1.\tau_1] \vdash_{\Sigma} e : \tau_2$, *and*
- $\mathit{Weaker}_{\Sigma}(FTV(\Lambda, \tau_1) \cap \mathit{dom}(\Sigma), 0)$,

then there exists $\Sigma' \leq_1 \Sigma$ *such that* $\Lambda; \Gamma \vdash_{\Sigma'} [v/x]e : \tau_2$.

Proof Sketch: This lemma must also be generalized to allow a proof by structural induction on the type derivation of e. In general, the strength context for the second assumption is of the form $(\Sigma, \Sigma_2) + c$, and for the conclusion, $(\Sigma', \Sigma_2) + c$. The strength context Σ_2 accounts for the local type variables in the ($\mathrm{LET_D}$) case, which cannot be allowed to decrease. The constant c generalizes the constant 1 or -1 added in the ($\mathrm{APP_D}$) and ($\mathrm{LAM_D}$) cases. Because of the generalization, the only interesting case is when $e = x$, which is proved as an instance of the following Type Substitution lemma and Weakening. □

Lemma (Type Substitution) *If*

- $\Lambda; \Gamma \vdash_{\Sigma,\Sigma_1} e : \tau$,
- $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$,
- *for all* $1 \leq i \leq n$, $\mathit{SWeaker}_{(\Sigma,\Sigma_2)+c}(FTV(\tau_i), \Sigma_1(\alpha_i))$,
- $\mathit{Weaker}_{\Sigma}(FTV(\Lambda, \Gamma, \tau) \cap \mathit{dom}(\Sigma), 0)$,

then there exists $\Sigma' \leq_1 \Sigma$ *such that* $S(\Lambda); S(\Gamma) \vdash_{(\Sigma',\Sigma_2)+c} e : S(\tau)$.

Proof Sketch: As with Value Substitution, the strength contexts are generalized so that the resulting lemma may be proved by structural induction on the type derivation. In particular, the first assumption uses $(\Sigma, \Sigma_1, \Sigma_0) + c'$, and the conclusion uses $(((\Sigma', \Sigma_2) + c), \Sigma_0) + c'$.

The ($\mathrm{VAR_D}$) base case is then proved by finding an appropriate Σ' such that the required instantiation holds. Similarly, the ($\mathrm{REF_D}$), ($\mathrm{APP_D}$), and ($\mathrm{LET_D}$) cases require calculating an an appropriate Σ' such that the side conditions hold, as well as using the Ground Type Substitution lemma to eliminate some type variables from consideration. The remaining cases follow easily. □

There are several other lemmas to strengthen and weaken type derivations. The previously mentioned Ground Type Substitution proves that any type derivations are stable when type variables are consistently replaced by ground types. Strengthening states that unused variables may be discarded from the variable type assumption and strength context. And Weakening shows that extra assumptions are allowed in the location and variable type assumptions and the strength context, and that the finite strength numbers may be safely decreased. In particular, $\lambda\Sigma$ does *not* allow infinite strengths to be decreased arbitrarily. For example,

$$\cdot;\cdot \vdash_{[\alpha \mapsto c, \beta \mapsto \infty]} \mathit{fn}\ f \Rightarrow \mathit{fn}\ x \Rightarrow f\,x : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

only if $c = \infty$ or $c \leq 2$.

5. Algorithmic Formalisms

The implementation of SML/NJ does not use this motivational formalism. While similar, one of its core ideas is the use of an *occurrence* which approximates the surrounding syntactic context of a subexpression. The "top-level" occurrence is named *Root*, and the mappings on occurrences are named *Rator*$(\cdot)$, *Rand*$(\cdot)$, *Abs*$(\cdot)$, and *Let*$(\cdot)$, on application functions, application arguments, abstraction bodies, and *let*-bound expressions, respectively. Various versions of SML/NJ include different fields in occurrences, and the remainder of this paper discusses the two primary ones, while other features not used are discussed in Section 6.

The Systems $\lambda\Sigma^-$ and $\lambda\Psi^-$

Instead of incrementing and decrementing every strength in the context at every abstraction and application, a single offset can be used. This offset, called the ***abstraction depth***, is the first field of an occurrence. Thus, we can split the strength context Σ into a strength context Ψ and an abstraction depth a such that $\Sigma = \Psi - a$, and the typing rules can be written leave Ψ fixed. The abstraction depth of a given subsexpression approximates the number of times that subexpression is applied in the whole expression. The only effect of making such a change is an increase in efficiency.

In order to motivate the other field used here, we digress temporarily. A more restrictive version of the typing rule for applications would be

$$\frac{\Lambda;\Gamma \vdash_{\Sigma+1} e_1 : \tau_2 \rightarrow \tau \qquad \Lambda;\Gamma \vdash_{\Sigma} e_2 : \tau_2}{\Lambda;\Gamma \vdash_{\Sigma} e_1\ e_2 : \tau} \quad \textit{if } \textit{Weaker}_{\Sigma}(FTV(\Lambda,\Gamma,\tau_2),0) \qquad (\mathrm{APP}'_{\mathrm{D}})$$

The type system $\lambda\Sigma^-$ resulting from replacing ($\mathrm{APP}_{\mathrm{D}}$) with ($\mathrm{APP}'_{\mathrm{D}}$) is sound since it is a (strict) subsystem of the original.

Surprisingly, however, the Value Substitution lemma does *not* hold in $\lambda\Sigma^-$. For example, let

$$\begin{aligned} v &= \textit{fn } a \Rightarrow a()()() \\ e &= \textit{fn } y \Rightarrow \textit{let } u{=}\textit{ref } y \textit{ in } x \end{aligned}$$

so that they are assigned non-critical types

$$\bullet;\bullet \vdash_{[\alpha\mapsto\infty]} v : \textit{unit}\rightarrow\textit{unit}\rightarrow\textit{unit}\rightarrow\alpha$$

$$\bullet;[x : \textit{unit}\rightarrow\textit{unit}\rightarrow\textit{unit}\rightarrow\alpha] \vdash_{[\alpha\mapsto\infty,\beta\mapsto 1]} e : \beta\rightarrow(\textit{unit}\rightarrow\textit{unit}\rightarrow\textit{unit}\rightarrow\alpha)\rightarrow\alpha$$

but the result of substitution is typed as

$$\bullet;\bullet \vdash_{[\alpha\mapsto\infty,\beta\mapsto 0]} [v/x]e : \beta\rightarrow(\textit{unit}\rightarrow\textit{unit}\rightarrow\textit{unit}\rightarrow\alpha)\rightarrow\alpha$$

where a non-critical strength has been lowered. As a result, the soundness of this system cannot be shown directly using the same style of proof as in Section 4.

Use of the second component of the occurrence, the ***maximum weakness*** w, allows the side condition on ($\mathrm{APP}'_{\mathrm{D}}$) to be replaced by a check at instantiation. For this system, there is no reason to prefer this alternative, although some motivation will be given for other systems. The maximum weakness is an upper bound on finite strengths in Ψ. The mappings on occurrences (a, w) are then defined by

$$\begin{aligned} \textit{Rand}(a,w) &= (a, \min(a,w)) \\ \textit{Rator}(a,w) &= (a-1, w) \\ \textit{Abs}(a,w) &= (a+1, w) \\ \textit{Let}(a,w) &= (a, \infty) \\ \textit{Root} &= (0, \infty) \end{aligned}$$

A type judgment $\Lambda; \Gamma \vdash_{\Psi; a, w} e : \tau$ reads "With strength context Ψ, at the occurrence containing the abstraction depth a and maximum weakness w, and given the location type assignment Λ and variable type assignment Γ, e has type τ." Such a judgment is well-formed if $dom(\Psi) \supseteq FTV(\Lambda, \Gamma, \tau)$. Derivability in $\lambda\Psi^-$ is defined by the following rules.

$$\frac{\vdash_{\Psi - a} \Gamma(x) \succeq \tau}{\Lambda; \Gamma \vdash_{\Psi; a, w} x : \tau} \quad \mathit{if}\ Weaker_{\Psi}(FTV(\Lambda, \Gamma, \tau), w) \qquad (\mathrm{VAR_A})$$

$$\frac{\Lambda(l) = \tau}{\Lambda; \Gamma \vdash_{\Psi; a, w} l : \tau\ ref} \quad \mathit{if}\ Weaker_{\Psi}(FTV(\Lambda, \Gamma), w) \qquad (\mathrm{LOC_A})$$

$$\Lambda; \Gamma \vdash_{\Psi; a, w} () : unit \quad \mathit{if}\ Weaker_{\Psi}(FTV(\Lambda, \Gamma), w) \qquad (\mathrm{UNIT_A})$$

$$\frac{\Lambda; \Gamma \vdash_{\Psi; Rand(a, w)} e : \tau}{\Lambda; \Gamma \vdash_{\Psi; a, w} ref\, e : \tau\ ref} \quad \mathit{if}\ Crit_{\Psi - a}(FTV(\tau)) \qquad (\mathrm{REF_A})$$

$$\frac{\Lambda; \Gamma \vdash_{\Psi; a, w} e : \tau\ ref}{\Lambda; \Gamma \vdash_{\Psi; a, w} !e : \tau} \qquad (!_\mathrm{A})$$

$$\frac{\Lambda; \Gamma \vdash_{\Psi; a, w} e_1 : \tau\ ref \qquad \Lambda; \Gamma \vdash_{\Psi; a, w} e_2 : \tau}{\Lambda; \Gamma \vdash_{\Psi; a, w} e_1 := e_2 : unit} \qquad (:=_\mathrm{A})$$

$$\frac{\Lambda; \Gamma \vdash_{\Psi; Rator(a, w)} e_1 : \tau_2 \rightarrow \tau \qquad \Lambda; \Gamma \vdash_{\Psi; Rand(a, w)} e_2 : \tau_2}{\Lambda; \Gamma \vdash_{\Psi; a, w} e_1\ e_2 : \tau} \qquad (\mathrm{APP_A})$$

$$\frac{\Lambda; \Gamma[x : \tau_1] \vdash_{\Psi; Abs(a, w)} e : \tau_2}{\Lambda; \Gamma \vdash_{\Psi; a, w} fn\ x \Rightarrow e : \tau_1 \rightarrow \tau_2} \quad \mathit{if}\ x \notin dom(\Gamma) \qquad (\mathrm{LAM_A})$$

$$\frac{\begin{array}{c}\Lambda; \Gamma \vdash_{\Psi.\Psi'; Let(a, w)} e_1 : \tau_1 \\ \Lambda; \Gamma[x : \forall\Psi' - a.\tau_1] \vdash_{\Psi; a, w} e_2 : \tau_2\end{array}}{\Lambda; \Gamma \vdash_{\Psi; a, w} let\ x{=}e_1\ in\ e_2 : \tau_2} \quad \mathit{if}\ \begin{array}{l}x \notin dom(\Gamma),\ NonCrit_{\Psi' - a}(dom(\Psi')),\ and \\ Weaker_{\Psi}((FTV(\tau_1) \cap dom(\Psi)) - FTV(\Lambda, \Gamma), a)\end{array} \qquad (\mathrm{LET_A})$$

Only the abstraction depth is incremented and decremented in ($\mathrm{APP_A}$) and ($\mathrm{LAM_A}$). But, the strength context offset by the abstraction depth is now used in ($\mathrm{VAR_A}$), ($\mathrm{REF_A}$), and ($\mathrm{LET_A}$).

Since $Let(a, w) = (a, w)$, ($\mathrm{LET_A}$) places no upper bound on the strengths of the type variables in Ψ'. The strengths in Ψ are still bound by the maximum weakness w in the type derivation of e_2. If instead $Let(a, w) = (a, w)$, then there would be no generalization in expressions such as $f(let\ x{=}fn\ z \Rightarrow ref\ z\ in\ e)$. Similarly, the top-level occurrence *Root* also has an infinite maximum weakness, so that no extra constraints are placed on strengths.

By structural induction, it is easy to show that

Lemma (Maximum Weakness – $\lambda\Psi^-$) *If* $\Lambda; \Gamma \vdash_{\Psi; a, w} e : \tau$*, then* $Weaker_{\Psi}(FTV(\Lambda, \Gamma, \tau), w)$.

In particular, this implies that the side condition of ($\mathrm{APP'_D}$) is satisfied by this system, also. Thus it is easy to show by induction that

Lemma ($\lambda\Sigma^-$ Contains $\lambda\Psi^-$) *If* $\Lambda;\Gamma \vdash_{\Psi;a,w} e : \tau$, *then* $\Lambda;\Gamma \vdash_{\Psi-a} e : \tau$.

In fact, these two systems are equivalent, in the sense that they admit the same derivations at the "top-level". This follows from

Lemma ($\lambda\Psi^-$ Contains $\lambda\Sigma^-$) *If* $\Lambda;\Gamma \vdash_{\Psi-a} e : \tau$, *and* $Weaker_\Psi(FTV(\Lambda,\Gamma,\tau), w)$, *then* $\Lambda;\Gamma \vdash_{\Psi;a,w} e : \tau$

which is proved by structural induction, and using Ground Type Substitution.

The System $\lambda\Psi$

The SML/NJ implementation is not as restrictive in its use of the maximum weakness. The following changes result in a system, $\lambda\Psi$, more like $\lambda\Sigma$ and the implementation.

$$\frac{\vdash_{\Psi-a} \Gamma(x) \succeq \tau}{\Lambda;\Gamma \vdash_{\Psi;a,w} x : \tau} \quad \textit{if } Weaker_\Psi(FTV(\tau), w) \qquad (\mathrm{VAR}'_\mathrm{A})$$

$$\frac{\Lambda(l) = \tau\ ref}{\Lambda;\Gamma \vdash_{\Psi;a,w} l : \tau} \qquad (\mathrm{LOC}'_\mathrm{A})$$

$$\Lambda;\Gamma \vdash_{\Psi;a,w} () : unit \qquad (\mathrm{UNIT}'_\mathrm{A})$$

For this system, it might be argued that the side condition of $(\mathrm{VAR}'_\mathrm{A})$ is more efficient than that of $(\mathrm{APP}_\mathrm{D})$ since instantiation must examine the strengths of some of the type variables of τ anyway.

This system is strictly less conservative than $\lambda\Psi^-$ and $\lambda\Sigma^-$, but is incomparable with $\lambda\Sigma$. For example, if

$$e = (fn\ z \Rightarrow z)(fn\ a \Rightarrow let\ x{=}fn\ y \Rightarrow ref\ a\ in\ fn\ b \Rightarrow ref\ b)$$

then in $\lambda\Psi$ we have $.;. \vdash_{[\alpha\mapsto 2,\beta\mapsto 0];Root} e : \alpha\rightarrow\beta\rightarrow\beta\ ref$. But, in $\lambda\Sigma$, the finite strengths of the argument must be critical, so α is at most 0. And if

$$e = (fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow z)(z{:=}fn\ a \Rightarrow ref\ a)())(ref\,(fn\ a \Rightarrow ref\ a))$$

then in $\lambda\Sigma$ we have $.;. \vdash_{[\alpha\mapsto 0]} e : \alpha\rightarrow\alpha\ ref$. But, in $\lambda\Psi$, α can only have strength -1 at the occurrence *Root*. It can be shown, however, that an expression typable in $\lambda\Sigma$ is also typable in $\lambda\Psi$ at the top-level occurrence by only lowering critical strengths.

As a result, the soundness of this system is still open, as is its stability under substitution. One factor that complicates proofs is that only very weak forms of a Maximum Weakness lemma hold as in the following.

Lemma (Maximum Weakness – $\lambda\Psi$) *If* $\Lambda;\Gamma \vdash_{\Psi;a,w} e : \tau_1\rightarrow\cdots\rightarrow\tau_n$, *then* $Weaker_\Psi(FTV(\tau_n), w)$.

The Systems $\lambda\Sigma^+$ and $\lambda\Psi^+$

The conservative approximation at application in all of the previous systems is overly conservative. As motivated, the finite strengths of the argument type must be critical, because the function may, in turn, apply its argument to other arguments, possibly creating a reference cell. If the argument is purely functional, however, this is impossible. But by using the same strength context (modulo a constant offset) to type both the function and argument, this cannot always be detected. For example, in $\lambda\Sigma$,

$$\bullet;\bullet \vdash_{[\alpha\mapsto 0, \beta\mapsto\infty]} (\mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow \mathit{ref}\ x)(\mathit{fn}\ a \Rightarrow a) : \beta \to (\alpha \to \alpha\ \mathit{ref})$$

Here, the argument, the identity function, must be given a weak type to match the strength of the function domain.

The following more complicated application rule does not force the conservative approximation on the type variables of the argument which "could have been" of infinite strength.

$$\frac{\Lambda;\Gamma \vdash_{\Sigma+1} e_1 : \tau_2 \to \tau \qquad \Lambda;\Gamma \vdash_{\Sigma'} e_2 : \tau_2}{\Lambda;\Gamma \vdash_{\Sigma} e_1\ e_2 : \tau} \quad \begin{array}{l} \mathit{Weaker}_{\Sigma'}(FTV(\tau_2), 0),\ \mathit{and} \\ \mathit{if}\ \Sigma'(\alpha) \geq \Sigma(\alpha)\ \mathit{if}\ \alpha \in FTV(\tau_2) - FTV(\Lambda, \Gamma) \\ \Sigma'(\alpha) = \Sigma(\alpha)\ \mathit{otherwise} \end{array} \qquad (\mathrm{APP}''_{\mathrm{D}})$$

This effectively makes part of unification explicit in the formalism. The system $\lambda\Sigma^+$, using this application rule, is strictly less conservative than $\lambda\Sigma$. The soundness proof extends to this system with little modification only if a weakening rule such as the following is also included.

$$\frac{\Lambda;\Gamma \vdash_{\Sigma} e : \tau}{\Lambda;\Gamma \vdash_{\Sigma'} e : \tau} \quad \mathit{if}\ \Sigma' \leq \Sigma \qquad (\mathrm{WEAKEN}_{\mathrm{D}})$$

where $\Sigma' \leq \Sigma$ if Σ' is point-wise less than or equal to Σ. Because of the Weakening lemma, it would be sufficient for the side condition to state that Σ and Σ' aggree on all finite values in Σ. Without such a rule, the system is not stable under substitution, but still might be sound.

This weakening rule is only useful immediately preceding application, so the typing rules could be combined. This would restore the syntax-directedness of the system, but would further complicate the side conditions on the application rule.

The algorithmic system $\lambda\Psi^+$ can be defined similarly, and is the most SML/NJ-like system in this paper. Like $\lambda\Psi$, however, its soundness is still open.

6. Relation to SML/NJ

Some differences between SML/NJ and the formalisms are primarily syntactic. First, SML types correspond to the pairing of types and strength contexts. And unlike SML, the formalisms restrict the reference primitives to their fully applied forms. It would be equivalent to replace the inference rules for the reference primitives with the following

$$\begin{array}{rcl} \mathit{ref} & : & \forall[\alpha\mapsto 1].\alpha \to \alpha\ \mathit{ref} \\ ! & : & \forall[\alpha\mapsto\infty].\alpha\ \mathit{ref} \to \alpha \\ := & : & \forall[\alpha\mapsto\infty].\alpha\ \mathit{ref} \to \alpha \to \mathit{unit} \end{array}$$

in the variable type assumption, or as the equivalent axioms. The disadvantage of this alternative is that values must be given to the evaluation of partially applied primitives, which complicates the dynamic semantics and loosens the correspondence of the inference rules of the static and dynamic semantics. However, the algorithmic formalisms are slightly stronger than the implementation in that the type inference rules for ! and := do not use *Rand*(·) as does general application. This is safe since these primitives are known not to create any cells when partially applied.

The implementation implicitly uses ($\mathrm{WEAKEN}_{\mathrm{D}}$) when unifying types in the application case. It was omitted from the majority of the formalisms because its non-syntax-directed nature complicates proofs. Or, if combined with the application rule of each system, it would hinder comprehension. The systems are sound, and admit more types for expressions, but apparently do not type more expressions with its addition.

Sequencing, $e_1; e_2$, can be treated as syntactic sugar for *let* $z{=}e_1$ *in* e_2, where z is new. The type inference rule in the declarative framework would be

$$\frac{\Lambda; \Gamma \vdash_\Sigma e_1 : \tau_1 \qquad \Lambda; \Gamma \vdash_\Sigma e_2 : \tau_2}{\Lambda; \Gamma \vdash_\Sigma e_1; e_2 : \tau_2} \qquad (;_{\mathrm{D}})$$

In the declarative formalisms, either definition of sequences admits the same expressions to be typed, although the definition as a *let* expression allows more derivations.

Also, the implementation has three additional fields in the occurrence. The ***lambda depth*** is similar to the abstraction depth, but is not decremented in *Rator*(·). Its use seems to be to prevent function expressions from being given critical types (except when the function is used as an argument, of course). In later versions, the ***base*** field provides a simpler, but unsound, analysis which allows

```
    (let val x = ref (fn z => z) in fn y => x end) () : ('1a -> '1a) ref
```

It is unclear how either of these relate to the declarative formalisms. The *outer* field allows curried function applications to be treated somewhat like a single uncurried application, by using the same occurrence for each of its arguments. This corresponds to having a single application rule for typing multiple curried arguments at once, as in

$$\frac{\Lambda; \Gamma \vdash_{\Sigma+n} e_0 : \tau_1 \to \cdots \tau_n \to \tau \qquad \Lambda; \Gamma \vdash_\Sigma e_i : \tau_i}{\Lambda; \Gamma \vdash_\Sigma e_0\ e_1 \ldots e_n : \tau} \quad \begin{array}{l} \textit{if } \mathit{Weaker}_\Sigma(FTV(\tau_i), 0), \\ \textit{for all } 1 \le i \le n \end{array} \qquad (\mathrm{APP} - \mathrm{many}_{\mathrm{D}})$$

Because of side effects, the implementation effectively does not use the same strength context when type checking multiple antecedents of an inference rule. For example, when type checking the following expression, the implementation decrements the *same* strength counter for each application `f()`.

```
    fn a => let val f = fn b => fn c => ref a in f(); f(); f() end
    : '0a -> 'c -> '0a ref
```

However, the formalisms allow the expression to have the type $\alpha \to \gamma \to \alpha$ *ref*, with the expected strength, $\Sigma(\alpha) = 2$.

7. Comparisons with Other Related Systems

Weak polymorphism is often explained as it is here, as a generalization of Tofte's imperative type system, but this is not entirely correct. Tofte's system uses *two* inference rules for type checking *let* expressions. One generalizes *all* type variables when the *let*-bound expression is non-expansive. The other generalizes applicative type variables when the expression is expansive. If an expression of critical type were necessarily expansive, then the *let* type inference rule would subsume both cases, but this is not so. For example, in the declarative systems,

$$\bullet; \bullet \vdash_{\alpha \mapsto s} \mathit{fn}\ a \Rightarrow (\mathit{let}\ x = \mathit{ref}\ a\ \mathit{in}\ \mathit{fn}\ y \Rightarrow x)() : \alpha \to \alpha\ \mathit{ref}$$

only if $s \le 0$. Thus, the expression is of critical type, but non-expansive. Thus, we conjecture that restricting any of the formalisms to using only the strengths 0 and ∞,[3] and augmenting it with Tofte's non-expansive *let* type inference rule is strictly more powerful than Tofte's system.

[3]Where $0 - n = 0$, $0 + n = 0$, and $\infty - n = 0$, for any n.

Hoang, Mitchell, and Viswanathan [4] proved the soundness of a different type system based on weak types. They permit different strengths on separate instances of a type variable in a type as in

```
fn f => f nil : ('sa list -> 'sa) -> '(s − 1)a
```

for any finite strength s. The decremented strength of the function result reflects the single application in the function body. This generalization of the SML/NJ approach gives a more informative analysis of strengths. even for purely functional terms as above, which eliminates the need for the conservative approximation of strengths at function applications. As a result, they claim that their system is more general than that of SML/NJ and provide empirical evidence of this, but they lack a formalization of SML/NJ to prove the claim.

In their analysis of reference creation, both weak and imperative types label type variables with information. Another approach is to label type arrows with *effects*, an approximation of the change in the store. The static semantics then derives both a type and an effect for an expression, and generalization is then defined relative to those effects. This approach is taken by Damas [1], Leroy and Weis [6], Reynolds [9], Talpin and Jouvelot [10, 11], and Wright [13]. A slightly different approach is given by Leroy [5], where type arrows are labelled with types that may occur in references. He also provides a comparison of this approach with some of these others.

For a comparison of some of these systems to each other, see Wright [13]. Also refer to O'Toole [8] for a rough comparison of four systems, including MacQueen's.[4] In general, this approach appears to be more powerful than weak types, although existing systems are incomparable. Furthermore, the systems using this approach have simpler inference rules than those shown here for weak polymorphism. However, in practice the approach may be unwieldy, because of the size of the type arrow labels.

Weak polymorphic types may be examined with this approach as well. As a rough outline, the types are defined as

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \mathit{unit} \mid \tau \xrightarrow{E} \tau \\ \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \tau \end{array}$$

where E is a set of type variables with the restriction that in the type $\tau_1 \xrightarrow{E_1} \cdots \xrightarrow{E_{n+1}} \tau_n$, if $\alpha \in E_i$, then $\alpha \in E_{i+1}$. (Unfortunately, we see no obvious motivation for this restriction in this setting.) Then the type $\tau = \tau_1 \rightarrow \cdots \rightarrow \tau_n$ in the strength context Σ corresponds to the type $\tau_1 \xrightarrow{E_1} \cdots \xrightarrow{E_{n-1}} \tau_n$ paired with the effect E_0, where $E_i = \{\alpha | \alpha \in FTV(\tau), \Sigma(\alpha) \leq i\}$. The following table gives some examples of the correspondence to SML/NJ types:

`'0a ref`	$\equiv$	$\alpha\ \mathit{ref}, \{\alpha\}$
`'1a -> '1a ref`	$\equiv$	$\alpha \xrightarrow{\alpha} \alpha\ \mathit{ref}, \emptyset$
`'2a -> 'b -> '2a ref`	$\equiv$	$\alpha \xrightarrow{\alpha} \beta \xrightarrow{\alpha} \alpha\ \mathit{ref}, \emptyset$
`('2a -> '0c) -> '1b -> '2a`	$\equiv$	$(\alpha \rightarrow \alpha) \xrightarrow{\beta\gamma} \beta \xrightarrow{\alpha\beta\gamma} \alpha, \{\gamma\}$

Wright [14] suggests that all of these systems are too complex for a practical type system, particularly in combination with modules. He notes that generalization is always sound if restricted to values, and gives empirical evidence that this restriction is not a great sacrifice in programming flexibility.

[4] O'Toole incorrectly allows generalization of critical type variables in his formalization of weak polymorphism.

8. Conclusions and Future Work

We have motivated and described several formalisms of weak polymorphic types which are quite similar to that of SML/NJ. In particular, the algorithmic family of calculi closely model the details of the implementation. Most of these have been proven sound with respect to the standard call-by-value operational semantics. Naturally, any of these could be incorporated into SML/NJ, to restore proven soundness to its type system. But, since weak polymorphism is also used to type continuations and exceptions, it should be verified that extending the system with these features is sound, although no complications are expected. The soundness of the remaining formalisms, $\lambda\Psi$ and $\lambda\Psi^+$, should also be determined, since they are closely related to the implementation.

Despite the similarities to SML/NJ, detailed comparisons are still somewhat difficult because the implementation has a broader definition of occurrences, and it uses side effects. The $\lambda\Psi$ family of formalisms could be enriched with the more general definition of an occurrence to further study some of these details. We conjecture that using the lambda depth field allows more function expressions to be non-critically typed, but still does not allow the type preservation theorem to hold without using more critical strengths.

These systems should be systematically compared to the alternate formalization of weak types found in [4]. This would test the claim that their approach is strictly more general than that of SML/NJ. It also provides a potential method of proving the soundness of $\lambda\Psi$ and $\lambda\Psi^+$, should their system be more general than these two calculi.

Many of the side conditions in the type rules are complex. This is especially true of the more powerful application rules, which more closely model the implementation. Since simplicity is one goal of a type system, so that it can be easily understood by the programmer, this points to a fundamental problem of practicality for the approach of weak polymorphism.

The connection between type systems which label type variables and those which label type arrows should also be further explored. Since specific systems of these two approaches are generally incomparable in power, it may be worthwhile to somehow combine the ideas in one system. However, such a combination would likely result in types too cumbersome in practice.

Acknowledgments

Many thanks go to Robert Harper and Mark Lillibridge for their many helpful ideas. I would also like to thank Peter Lee and Mark Leone for their comments.

References

[1] L. Damas. *Type Assignment in Programming Languages*. Ph.D. Thesis, University of Edinburgh. April, 1985.

[2] Robert Harper. *A Simplified Account of Polymorphic References*. Unpublished manuscript. April, 1992.

[3] J. Roger Hindley, Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, London Mathematical Society Student Texts, Volume 1. 1986.

[4] My Hoang, John Mitchell, Ramesh Viswanathan. *Standard ML weak polymorphism and imperative constructs*. To appear in *Logic in Computer Science*. 1993.

[5] Xavier Leroy. *Polymorphic Typing of An Algorithmic Language*. Ph.D. Thesis, University Paris VII. Technical Report 1778, Institute National de Recherche en Informatique et en Automatique. October, 1992.

[6] Xavier Leroy, Pierre Weis. *Polymorphic type inference and assignment.* In *ACM Symposium on Principles of Programming Languages*, pages 291–302. 1991.

[7] Dave MacQueen. Source code for SML/NJ type inference algorithm.

[8] James William O'Toole, Jr. *Type Abstraction Rules for References: A Comparison of Four Which Have Achieved Notoriety.* Technical report MIT-LCS-TM-390, MIT. 1989.

[9] John C. Reynolds. *Syntactic Control of Interference, Part 2.* In *International Colloquium on Automata, Languages, and Programming*, pages 704–722. July, 1989.

[10] Jean-Pierre Talpin, Pierre Jouvelot. *The Type and Effect Discipline.* In *7th IEEE Symposium on Logic in Computer Science*. pp. 162–173. IEEE Comput. Soc. Press. 1992.

[11] Jean-Pierre Talpin, Pierre Jouvelot. *Polymorphic Type, Region and Effect Inference.* In *Journal of Functional Programming*, Vol. 2, No. 3. pp. 245–271. Cambridge University Press. 1992.

[12] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* Ph.D. Thesis, University of Edinborough. 1988.

[13] Andrew K. Wright. *Typing References by Effect Inference.* In *European Symposium on Programming*, Lecture Notes in Computer Science, volume 582. pp. 473–491. February, 1992.

[14] Andrew K. Wright *Polymorphism for Imperative Languages without Imperative Types.* Technical Report 93–200, Rice University. February, 1993.

[15] Andrew K. Wright, Matthias Felleisen. *A Syntactic Approach to Type Soundness.* Technical Report 91–160, Rice University. July, 1991.

Appendix – Proof

This appendix presents the proofs of the soundness theorem and lemmas for system $\lambda\Sigma$, as outlined in Section 4. They are presented in the order of their dependence.

Lemma 1 (Weakening) *If*

- $\Lambda; \Gamma \vdash_{\Sigma_0} e : \tau$,
- $\Sigma_0' \leq_\infty \Sigma_0$,
- $FTV(\Lambda', \Gamma') \subseteq dom(\Sigma_0, \Sigma_1)$,
- $dom(\Lambda) \cap dom(\Lambda') = \emptyset$, *and*
- $dom(\Gamma) \cap dom(\Gamma') = \emptyset$,

then $\Lambda, \Lambda'; \Gamma, \Gamma' \vdash_{\Sigma_0', \Sigma_1} e : \tau$.

Several subcases of this weakening lemma are used, and it is proved by straightforward structural induction on the type derivation.

Lemma 2 (α-Renaming Type Substitution) *If*

- $\Lambda; \Gamma \vdash_\Sigma e : \tau$,
- $S = [\alpha_1 \mapsto \alpha_1', \ldots, \alpha_n \mapsto \alpha_n']$,
- $dom(S) \subseteq dom(\Sigma)$, *and*

- $rng(S) \cap dom(\Sigma) = \emptyset$,

then $S(\Lambda); S(\Gamma) \vdash_{S(\Sigma)} e : S(\tau)$.

Lemma 3 (Ground Type Substitution) *If*

- $\Lambda; \Gamma \vdash_{\Sigma} e : \tau$,
- $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$,
- $dom(S) \subseteq dom(\Sigma)$, *and*
- $FTV(rng(S)) = \emptyset$,

then $S(\Lambda); S(\Gamma) \vdash_{\Sigma} e : S(\tau)$.

These two lemmas are special cases of type substitutions. The first one is used to show that the names of generalized variables can be α-renamed to avoid conflicts with other strength contexts. The second is used to eliminate unnecessary type variables from consideration. Both follow easily by structural induction on the type derivation. See the proof of Type Substitution for details of the (VAR_{D}) case.

Lemma 4 (Strengthening) *If*

- $\Lambda; \Gamma, \Gamma' \vdash_{\Sigma, \Sigma'} e : \tau$,
- $FTV(\Lambda, \Gamma, \tau) \subset dom(\Sigma)$, *and*
- $dom(\Gamma') \cap FV(e) = \emptyset$,

then $\Lambda; \Gamma \vdash_{\Sigma} e : \tau$.

This lemma could easily be generalized to strengthen the location type assumptions, as well, but this is unnecessary.

Proof: It is proved by simple structural induction on the type derivation, using Ground Type Substitution to eliminate any extra type variables occurring only in the mediating types in ($:=_{\text{D}}$), (APP_{D}), and (LET_{D}). For the most difficult case, (LET_{D}), inversion of the derivation provides a Σ_0 and τ' such that

$$\Lambda; \Gamma \vdash_{\Sigma, \Sigma_0, \Sigma'} e_1 : \tau' \qquad \Lambda; \Gamma[x : \forall \Sigma_0 . \tau'] \vdash_{\Sigma, \Sigma'} e_2 : \tau$$

$$NonCrit_{\Sigma_0}(dom(\Sigma_0)) \qquad Weaker_{\Sigma}(FTV(\tau') \cap dom(\Sigma) - FTV(\Lambda, \Gamma), 0)$$

Without loss of generality, assume by α-renaming on values, that $x \notin dom(\Gamma')$.

Induction cannot be used yet, since $FTV(\tau')$ is not necessarily a subset of $dom(\Sigma, \Sigma_0)$. So, define S to be a ground type substitution on the type variables in $dom(\Sigma') \cap FTV(\tau')$. Then, Ground Type Substitution gives

$$\Lambda; \Gamma \vdash_{\Sigma, \Sigma_0, \Sigma'} e_1 : S(\tau') \qquad \Lambda; \Gamma[x : S(\forall \Sigma_0 . \tau')] \vdash_{\Sigma, \Sigma'} e_2 : \tau$$

And $S(\forall \Sigma_0 . \tau') = \forall \Sigma_0 . S(\tau')$ since the domains of S and Σ_0 are non-intersecting. Induction then applies, and the conclusion follows by (LET_{D}). $\square$

Lemma 5 (Type Substitution) *If*

- $\Lambda; \Gamma \vdash_{(\Sigma, \Sigma_1, \Sigma_0) + c'} e : \tau$,
- $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$,

- *for all* $1 \leq i \leq n$, $SWeaker_{(\Sigma,\Sigma_2)+c}(FTV(\tau_i), \Sigma_1(\alpha_i))$, *and*
- $Weaker_{\Sigma}(FTV(\Lambda, \Gamma, \tau) \cap dom(\Sigma), 0)$,

then there exists $\Sigma' \leq_1 \Sigma$ *such that* $S(\Lambda); S(\Gamma) \vdash_{(((\Sigma',\Sigma_2)+c),\Sigma_0)+c'} e : S(\tau)$.

Proof: This is proved by structural induction on the typing derivation. The (LOC_D) and (UNIT_D) cases hold trivially with the definition $\Sigma' = \Sigma$.

The (VAR_D) case requires careful treatment of the domains of substitutions to show that instantiation is stable under type substitution. Invertion of the type derivation states that the instantiation $\vdash_{(\Sigma,\Sigma_0,\Sigma_1)+c_2} \Gamma(x) \succeq \tau$ holds. Let

$$S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$$

where $\alpha_1, \ldots, \alpha_k \notin BTV(\Gamma(x))$, and where $\alpha_{k+1}, \ldots, \alpha_n \in BTV(\Gamma(x))$. By the definition of the instantiation relation,

$$\Gamma(x) = \forall[\alpha_{k+1} \mapsto s_1, \ldots, \alpha_{k+m} \mapsto s_m].\tau'$$

where $k + m \geq n$, and there exists a types $\tau_1', \ldots, \tau_m'$ such that defining the type substitution

$$S_1 = [\alpha_{k+1} \mapsto \tau_1', \ldots, \alpha_{k+m} \mapsto \tau_m']$$

implies $S_1(\tau') = \tau$. In particular, choose S_1 so that, for $1 \leq j \leq m$, if $\alpha_{k+j} \notin FTV(\tau')$, then τ_j' is ground, so that $FTV(rng(S)) \subseteq FTV(\tau)$. In addition, define the following type substitutions, where $\alpha_1'', \ldots, \alpha_n''$ are new:

- $S' = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k]$
- $S_2 = [\alpha_1'' \mapsto S(\tau_1'), \ldots, \alpha_m'' \mapsto S(\tau_m')]$
- $S_3 = [\alpha_{k+1} \mapsto \alpha_1'', \ldots, \alpha_{k+m} \mapsto \alpha_{k+m}]$

Thus, $S(\Gamma(x)) = \forall[\alpha_1'' \mapsto s_1, \ldots, \alpha_m'' \mapsto s_m].S'(S_3(\tau'))$, and $S_2(S'(S_3(\tau'))) = S(S_1(\tau')) = S(\tau)$. Now define Σ' so that for all $\alpha \in dom(\Sigma') = dom(\Sigma)$,

$$\Sigma'(\alpha) = \begin{cases} \Sigma(\alpha) + \min(0, -c_1) & \text{if } \alpha \in FTV(rng(S)) \text{ and } \Sigma(\alpha) < \infty \\ \Sigma(\alpha) & \text{otherwise} \end{cases}$$

By the last assumption, and since $FTV(rng(S)) \subset FTV(\tau)$, then $\Sigma' \leq_1 \Sigma$. And since by instantiation, for all $1 \leq j \leq m$, $SWeaker_{(\Sigma,\Sigma_1,\Sigma_0)+c_2}(FTV(\tau_j'), s_j)$, the third assumption implies that, for all $1 \leq j \leq m$, $SWeaker_{(((\Sigma',\Sigma_2)+c_1),\Sigma_0)+c_2}(FTV(S(\tau_j')), s_j)$. Thus, the desired instantiation holds,

$$\vdash_{(((\Sigma',\Sigma_2)+c_1),\Sigma_0)+c_2} S(\Gamma(x)) \succeq S(\tau)$$

and the conclusion holds by (VAR).

The ($!_\text{D}$) and (LAM_D) cases follow simply by induction. Both the ($:=_\text{D}$) and (APP_D) cases must also use Ground Type Substitution, while in the (REF_D), (APP_D), and (LET_D) cases, an appropriate Σ' must be calculated to satisfy the side conditions, as in the (VAR_D) case. For example, in the (APP_D) case, inversion gives a τ' such that

$$\Lambda; \Gamma \vdash_{(\Sigma,\Sigma_1,\Sigma_0)+c_2+1} e_1 : \tau' \to \tau \qquad \Lambda; \Gamma \vdash_{(\Sigma,\Sigma_1,\Sigma_0)+c_2} e_2 : \tau' \qquad Weaker_{(\Sigma,\Sigma_1)+c_2}(FTV(\tau'), 0)$$

To allow the last assumption to hold inductively, the type variables occurring only in the mediating type τ' must be removed. So, define a ground type substitution S' over the type variables in $dom(\Sigma') \cap FTV(\tau') - FTV(\Lambda, \Gamma, \tau)$. By Ground Type Substitution,

$$\Lambda; \Gamma \vdash_{(\Sigma,\Sigma_1,\Sigma_0)+c_2+1} e_1 : S'(\tau') \to \tau \qquad \Lambda; \Gamma \vdash_{(\Sigma,\Sigma_1,\Sigma_0)+c_2} e_2 : S'(\tau')$$

Since $\mathit{Weaker}_{\Sigma}(S'(\tau') \cap \mathit{dom}(\Sigma), 0)$, induction proves that there exists $\Sigma'' \leq_1 \Sigma$ and $\Sigma''' \leq_1 \Sigma$ such that

$$S(\Lambda); S(\Gamma) \vdash_{(((\Sigma'', \Sigma_2)+c_1), \Sigma_0)+c_2+1} e_1 : S(S'(\tau') \rightarrow \tau) \qquad S(\Lambda); S(\Gamma) \vdash_{(((\Sigma''', \Sigma_2)+c_1), \Sigma_0)+c_2} e_2 : S(S'(\tau'))$$

and let $\Sigma'''' = \min(\Sigma'', \Sigma''')$. Now the side condition of $(\mathrm{APP_D})$ must be satisfied. To this end, define Σ' so that for all $\alpha \in \mathit{dom}(\Sigma') = \mathit{dom}(\Sigma)$,

$$\Sigma'(\alpha) = \begin{cases} \Sigma''''(\alpha) + \min(0, -c_1) & \text{if } \alpha \in \mathit{FTV}(S'(\tau')) \text{ and } \Sigma''''(\alpha) < \infty \\ \Sigma''''(\alpha) & \text{otherwise} \end{cases}$$

So that by the original side condition and the third assumption, the new side condition holds:

$$\mathit{Weaker}_{(((\Sigma', \Sigma_2)+c_1), \Sigma_0)+c_2}(\mathit{FTV}(S(S'(\tau'))), 0)$$

Also, by the original side condition and the last assumption, $\Sigma' \leq_1 \Sigma''''$. So, the conclusion holds by Weakening and the $(\mathrm{APP_D})$ rule. □

Lemma 6 (Value Substitution) *If*

- $\Lambda; . \vdash_{\Sigma, \Sigma_1} v : \tau_1$,
- $\Lambda; \Gamma[x : \forall \Sigma_1 . \tau_1] \vdash_{(\Sigma, \Sigma_2)+c} e : \tau_2$, *and*
- $\mathit{Weaker}_{\Sigma}(\mathit{FTV}(\Lambda, \tau_1) \cap \mathit{dom}(\Sigma), 0)$,

then there exists $\Sigma' \leq_1 \Sigma$ *such that* $\Lambda; \Gamma \vdash_{(\Sigma', \Sigma_2)+c} [v/x]e : \tau_2$.

Proof: The proof is by structural induction on the type derivation for e. The $(\mathrm{LOC_D})$, $(\mathrm{UNIT_D})$, and when $e \neq x$, $(\mathrm{VAR_D})$ *cases follow from Strengthening with the definition* $\Sigma' = \Sigma$. *When* $e = x$, then $\vdash_{(\Sigma, \Sigma_2)+c} \forall \Sigma_1 . \tau_1 \succeq \tau_2$ follows by inversion. By the definition of instantiation, there is a type substitution S such that $S(\tau_1) = \tau_2$, and for all $\alpha' \in \mathit{dom}(S)$, $S\mathit{Weaker}_{(\Sigma, \Sigma_2)+c}(\mathit{FTV}(S(\alpha')), \Sigma_1(\alpha'))$. Also, $S(\Lambda) = \Lambda$. The conclusion holds by Type Substitution and Strengthening.

The $(\mathrm{REF_D})$, $(!_\mathrm{D})$, and $(\mathrm{LAM_D})$ cases holds by induction. The $(:=_\mathrm{D})$ and $(\mathrm{APP_D})$ cases hold by induction and Weakening. In the $(\mathrm{LET_D})$ case, inversion states that there exists a τ' and Σ_0 such that

$$\Lambda; \Gamma[x : \forall \Sigma_1 . \tau_1] \vdash_{(\Sigma, \Sigma_2, \Sigma_0)+c} e_1 : \tau' \qquad \Lambda; \Gamma[x : \forall \Sigma_1 . \tau_1, y : \forall \Sigma_0 . \tau'] \vdash_{(\Sigma, \Sigma_2)+c} e_2 : \tau_2$$

$$\mathit{NonCrit}_{\Sigma_0}(\mathit{dom}(\Sigma_0)) \qquad \mathit{Weaker}_{(\Sigma', \Sigma_2)+c}(\mathit{FTV}(\tau') \cap \mathit{dom}(\Sigma, \Sigma_2) - \mathit{FTV}(\Lambda, \Gamma[x : \forall \Sigma_1 . \tau_1]), 0)$$

By α-Renaming Type Substitution, we can assume that the type variables in Σ_0 do not clash with those in Σ_1. So, by induction, there exists $\Sigma'' \leq_1 \Sigma$ and $\Sigma''' \leq_1 \Sigma$ such that

$$\Lambda; \Gamma \vdash_{(\Sigma'', \Sigma_2, \Sigma_0)+c} [v/x]e_1 : \tau' \qquad \Lambda; \Gamma[y : \forall \Sigma_0 . \tau'] \vdash_{(\Sigma''', \Sigma_2)+c} [v/x]e_2 : \tau_2$$

Let $\Sigma'''' = \min(\Sigma'', \Sigma''')$. In order to satisfy the last side condition of $(\mathrm{LET_D})$, define Σ' so that for all $\alpha \in \mathit{dom}(\Sigma') = \mathit{dom}(\Sigma)$,

$$\Sigma'(\alpha) = \begin{cases} \min(\Sigma''''(\alpha), -c) & \text{if } \alpha \in \mathit{FTV}(\tau_1) \text{ and } \Sigma''''(\alpha) < \infty \\ \Sigma''''(\alpha) & \text{otherwise} \end{cases}$$

By the last assumption, $\Sigma' \leq_1 \Sigma$, and $\mathit{Weaker}_{\Sigma'+c}(\mathit{FTV}(\forall \Sigma_1 . \tau_1), 0)$. So,

$$\mathit{Weaker}_{(\Sigma', \Sigma_2)+c}(\mathit{FTV}(\tau') \cap \mathit{dom}(\Sigma, \Sigma_2) - \mathit{FTV}(\Lambda, \Gamma), 0)$$

and the conclusion holds by Weakening and $(\mathrm{LET_D})$. □

Theorem 1 (Type Preservation Under Evaluation) *If*

- $\mu \vdash e \Longrightarrow v, \mu'$,
- $\Lambda; \bullet \vdash_\Sigma e : \tau$,
- $\vdash_{\Sigma+c} \mu : \Lambda$,
- $c \leq 0$, *and*
- $Crit_{\Sigma+c}(FTV(\Lambda))$,

then there exists Λ_0 *and* Σ_0 *such that*

- $\Sigma_0 + c \leq_1 \Sigma + c$
- $\Lambda, \Lambda_0; \bullet \vdash_{\Sigma_0} v : \tau$,
- $\vdash_{\Sigma_0+c} \mu' : \Lambda, \Lambda_0$, *and*
- $Crit_{\Sigma_0+c}(FTV(\Lambda_0))$

Proof: The proof is by structural induction on the evaluation derivation. The constant c corresponds exactly with the abstraction depth of the algorithmic formalisms, so the strength context $\Sigma + c$ is that of the top-level. Note that the type judgments in the assumption and conclusion use strength contexts that are *not* adjusted by c. Since the location type assumption and strength context are extended and weakened for each sub-derivation, the weakening lemma is needed to prove the assumptions of many induction cases.

The (VAL) case holds with $\Lambda_0 = \bullet$ and $\Sigma_0 = \Sigma$, since $\mu' = \mu$.

For the (ALLOC) case, inversion of the first two assumptions gives

$$\mu \vdash e \Longrightarrow v, \mu_1 \qquad \mu' = \mu_1[l \mapsto v] \qquad \Lambda; \bullet \vdash_\Sigma e : \tau \qquad Crit_\Sigma(FTV(\tau))$$

So, induction provides a Λ_1 and Σ_0 such that

$$\Sigma_0 + c \leq_1 \Sigma + c \qquad \Lambda, \Lambda_1; \bullet \vdash_{\Sigma_0} v : \tau \qquad \vdash_{\Sigma_0+c} \mu_1 : \Lambda, \Lambda_1 \qquad Crit_{\Sigma_0+c}(FTV(\Lambda_1))$$

Defining $\Lambda_0 = \Lambda_1[l : \tau\ ref]$, then $\vdash_{\Sigma_0+c} \mu' : \Lambda, \Lambda_0$. And $\Lambda, \Lambda_0; \bullet \vdash_{\Sigma_0} l : \tau\ ref$ holds by (LOC). Since $FTV(\Lambda_0) = FTV(\Lambda_1, \tau)$, the conclusion then holds.

In the (CONT) case, inversion gives

$$\Lambda; \bullet \vdash_\Sigma e : \tau\ ref \qquad \mu \vdash e \Longrightarrow l, \mu'$$

where $v = \mu'(l)$. Induction then proves that there exists a Λ_0 and Σ_1 such that

$$\Sigma_1 + c \leq_1 \Sigma + c \qquad \Lambda, \Lambda_0; \bullet \vdash_{\Sigma_1} l : \tau\ ref \qquad \vdash_{\Sigma_1+c} \mu' : \Lambda, \Lambda_0 \qquad Crit_{\Sigma_1+c}(FTV(\Lambda_0))$$

So, $\Lambda, \Lambda_0(l) = \tau\ ref$. Then, by the definition of type matching and Strengthening, $\Lambda, \Lambda_0; \bullet \vdash_{\Sigma_1+c} v : \tau$. Define Σ_0 so that for all $\alpha \in dom(\Sigma_0) = dom(\Sigma)$,

$$\Sigma_0(\alpha) = \begin{cases} \Sigma_1(\alpha) + c & \text{if } \alpha \in FTV(\Lambda, \Lambda_0) \\ \Sigma_1(\alpha) & \text{otherwise} \end{cases}$$

Since $Crit_{\Sigma_1+c}(FTV(\Lambda, \Lambda_0))$, then $\Sigma_0 + c \leq_1 \Sigma + c$. The type judgments of the conclusion then follow by Strengthening and Weakening, since $c \leq 0$.

The (UPD), (APPLY), and (BIND) cases are similar, with a pattern of induction followed by weakening. The latter two also use Value Substitution to allow induction on the result of substitution. Because of the similarities, only the (APPLY) case is given here.

For (APPLY), inversion gives a τ_2 such that

$$\mu \vdash e_1 \Longrightarrow fn\ x \Rightarrow e_1', \mu_1 \qquad \mu_1 \vdash e_2 \Longrightarrow v_2, \mu_2 \qquad \mu_2 \vdash [v_2/x]e_1' \Longrightarrow v, \mu'$$

$$\Lambda; \bullet \vdash_{\Sigma+1} e_1 : \tau_2 \rightarrow \tau \qquad \Lambda; \bullet \vdash_{\Sigma} e_2 : \tau_2 \qquad Weaker_{\Sigma}(FTV(\tau_2), 0)$$

Induction on the derivation involving e_1 shows that there exists a Λ_1 and Σ_1 such that

$$\Sigma_1 + 1 + (c-1) \leq_1 \Sigma + 1 + (c-1) \qquad \Lambda, \Lambda_1; \bullet \vdash_{\Sigma_1+1} fn\ x \Rightarrow e_1' : \tau_2 \rightarrow \tau$$

$$\vdash_{\Sigma_1+1+(c-1)} \mu_1 : \Lambda, \Lambda_1 \qquad Crit_{\Sigma_1+1+(c-1)}(FTV(\Lambda_1))$$

Weakening proves $\Lambda, \Lambda_1; \bullet \vdash_{\Sigma_1} e_2 : \tau_2$ so that induction applies, giving a Λ_2 and Σ_2 such that

$$\Sigma_2 + c \leq_1 \Sigma_1 + c \qquad \Lambda, \Lambda_1, \Lambda_2; \bullet \vdash_{\Sigma_2} v_2 : \tau_2 \qquad \vdash_{\Sigma_2+c} \mu_2 : \Lambda, \Lambda_1, \Lambda_2 \qquad Crit_{\Sigma_2+c}(FTV(\Lambda_2))$$

Inversion on the function value, along with Weakening, proves

$$\Lambda, \Lambda_1, \Lambda_2; [x : \tau_2] \vdash_{\Sigma_2} e_1' : \tau \qquad \Lambda, \Lambda_1, \Lambda_2; \bullet \vdash_{\Sigma_2} v_2 : \tau_2$$

So, using the trivial generalization $\forall \bullet . \tau_2 = \tau_2$. Value Substitution proves that there exists $\Sigma_3 \leq_1 \Sigma_2$ such that $\Lambda, \Lambda_1, \Lambda_2; \bullet \vdash_{\Sigma_3} [v_2/x]e_1' : \tau$. Since $c \leq 0$, then $\Sigma_3 + c \leq_1 \Sigma_2 + c$, and induction applies again, giving a Λ_3 and Σ_0 such that

$$\Sigma_0 + c \leq_1 \Sigma_3 + c \qquad \Lambda, \Lambda_1, \Lambda_2, \Lambda_3; \bullet \vdash_{\Sigma_0} v : \tau \qquad \vdash_{\Sigma_0+c} \mu' : \Lambda, \Lambda_1, \Lambda_2, \Lambda_3 \qquad Crit_{\Sigma_0+c}(FTV(\Lambda_3))$$

The conclusion then holds with $\Lambda_0 = \Lambda_1, \Lambda_2, \Lambda_3$. □